MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A.I. LABORATORY

Artificial Intelligence January 1974 Memo No. 302

# A Relaxation Approach To Splitting In An Automatic Theorem Prover

by

Arthur J. Nevins

## Abstract

The splitting of a problem into subproblems often involves the same variable appearing in more than one of the subproblems. This makes these subproblems dependent upon one another since a solution to one may not qualify as a solution to another. A two stage method of splitting is described which first obtains solutions by relaxing the dependency requirement and then attempts to reconcile solutions to different subproblems. The method has been realized as part of an automatic theorem prover programmed in LISP which takes advantage of the procedural power that LISP provides. The program has had success with cryptarithmetic problems, problems from the blocks world, and has been used as a subroutine in a plane geometry theorem prover.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-70-A-0362-0003.

## 1. Introduction

The splitting of a problem into subproblems often involves the same variable appearing in more than one of the subproblems. This makes these subproblems dependent upon one another since a solution to one may not qualify as a solution to another. This paper describes a two stage method of splitting which (1) first solves the subproblems as if they were independent (i.e. completely relaxes the dependency requirement) and (2) then attempts to reconcile solutions to different subproblems. The advantages of this approach lie in the economy derived from making only a single pass at finding solutions and in the ability to use global knowledge about the solutions obtained in stage 1 to assist the search for a reconciliation of these solutions in stage 2.

A possible disadvantage of this method is that it could be difficult to decide when to stop searching for unconstrained solutions to a subproblem in stage 1, particularly when the set of such solutions might be quite large. In order to make this method work for an interesting class of problems, heuristics were employed to prevent the search taking too long in cases where the number of solutions is infinite or very large. In particular, every case to a split need not be handled during the relaxation stage; some may be postponed to the reconciliation stage where constraints arising from the other cases can be brought to bear. The method has been realized as part of an automatic theorem prover programmed in LISP. The program has had success with cryptarithmetic problems [9], problems from the blocks world

[13], and has been used as a subroutine in a plane geometry theorem prover [8].

The reconciliation stage bears a resemblance to constraint methods such as used in [3]. The reconciliation stage also can be thought of as an attempt to "debug" solutions which were obtained by first ignoring interactions between subproblems and therefore is related to [12]. The overall approach to splitting is related to methods used in [2] where the investigation of a subproblem resulted in the assignment of a type to a variable appearing in that subproblem. Although the present procedure is oriented more to problems which allow for explicit enumeration of solutions, it would not be adverse to the addition of a type theory mechanism.

The logical formalism underlying the program is presented in section 2 and some considerations of a semantic nature are discussed in section 3. The program also can make use of procedural knowledge as in [1], [2], [3], [4], [6], [11] and this facility is described in section 4. The relaxation and reconciliation stages are examined in sections 5 and 6 respectively. Section 7 will provide some illustrative examples of the program's performance.

## 2. The Logical Deduction Rules

The reader is referred to [10] for background material on predicate calculus theorem provers. It will be assumed that the negation of the goal is to be regarded as a true formula, all formulas are in prenex form, and all existential variables have

been replaced by skolem functions [10]. We adopt the convention that E and E' always represent expressions which have been made identical by the unification algorithm [10]. The theorem prover has as its logical basis the following 11 rules of inference:

R1. A problem is solved when it has been established that both the literals A' and ~A are true.

R2. Replace formula ~~A by A.

R3. Replace formula A∧B by A, B.

R4. Replace formula ~(A∨B) by ~A, ~B
(i.e. if one wishes to prove A∨B, then either prove A or prove B).

R5. Replace formula ~(A→B) by A, ~B
(i.e. if one wishes to prove A→B, then assume A and prove B).

R6. (Modus ponens) If it has been established that A' and A→B are true where A is a literal, then add B' to the set of true formulas.

R7. (Modus ponens) If it has been established that ~B' and A→B are true where B is an atomic formula, then add ~A' to the set of true formulas.

R8. (Modus ponens) If it has been established that B' and A→~B are true where B is an atomic formula, then add ~A' to the set of true formulas.

R9. (Reasoning by cases) If A∨B is true, then split A∨B into case A and case B. If A and B are independent (i.e. A and B have no variables in common), then break the problem into two subproblems by first

assuming A and then assuming B. The treatment then would be the same as in [7] where an intensive effort would be made to solve a case before resorting to an additional case analysis and where a mechanism for curbing an explosion of cases is employed. However, if A and B have a variable in common, then the case analysis is conducted as described in sections 5 and 6.

R10. Replace formula $\sim(A \wedge B)$ by $\sim A \vee \sim B$.

R11. (Equality relation) If $r = t$ and $A(r)$ are true where $A(r)$ is a literal in which the term r appears, then each occurrence of r in $A(r)$ is replaced by the term t. This treatment of equality is crude compared with other automatic theorem provers [5], [7] and most likely would inhibit the ability of the program to prove theorems in subjects like group theory or number theory. However, there is nothing about the present program which would prevent the introduction of the more sophisticated treatment of equality which this author employed in [7].

The above rules are the same as in [7] except as already noted with respect to R9 and R11. The control structure governing the execution of these rules also is the same as in [7]. A comparison of this system with resolution theorem proving [10] already has been presented in [7] and will not be repeated here. The main objective of the present paper is to present a better approach to splitting than was employed in [7].

## 3. Some Semantic Considerations

The present use of semantics bears a resemblance to the

"model" strategies which have been used by resolution theorem provers [10]. It is based upon the duality that is associated with different statements in the data base. For the statement ~A can be regarded as asserting that ~A is true (and hence A is false) or it can be regarded as asserting that we wish to prove A. From the point of view of the formal logic, it makes no difference which interpretation you choose. But from the point of view of the underlying semantics it makes all the difference in the world whether you believe A to be false or whether you believe that you actually can prove A. The commonly held belief that predicate calculus theorem provers reason only by "contradiction" is rooted in the notion that the theorem prover either does not employ semantics or else employs semantics which do not distinguish the denial of the statement to be proved from other statements in the data base.

The program divides statements into "facts" and "goals." Initially, all axioms are regarded as facts and the negation of the statement to be proved is regarded as a goal. Two statements are not allowed to interact with each other (as paired inputs to the same inference rule) unless one of the statements is either the initial goal or a descendant of the initial goal. The initial goal therefore comprises a set of support [14]. Of course, there is some flexibility as to what actually constitutes the statement to be proved. For example, the set of statements A, B, C, D, ~E could represent the initial data base as well as the set of statements A, B, ~((C∧D)→E) so far as the formal logic is concerned. However, the latter representation would be

preferred if semantic considerations indicated that the hypotheses C and D were specifically related to the goal of proving E. In the latter representation, C and D would be generated as descendants of the initial goal through the application of rule R5 followed by R3 whereas they would not be regarded as descendants of the initial goal in the former representation.

As a new statement is generated by a deduction rule, the program makes a determination as to whether the statement is a fact or a goal. In rule R9, a case is regarded as a goal if and only if the case has a negation sign at its front and the formula being split is a goal. The output of rule R10 is regarded as a goal if and only if the input to that rule is a goal. If ~(A→B) were a goal, rule R5 would result in the output A treated as a fact and the output ~B as a goal. For all the other deduction rules, an output is regarded as a goal if and only if one of the inputs is a goal and the output has a negation sign at its front. Thus, the output B' of rule R6 would be a fact if A' were a fact since A→B as the second input to R6 could not possibly be a goal. This separation into facts and goals can help to control interactions in the data base. For suppose ~A and ~B are goals and neither is an ancestor of the other; the program then normally would allow ~A and ~B to interact with each other but would not allow descendants of ~A to interact with descendants of ~B.

The symbol → has an important semantic role which has been neglected by resolution theorem provers. Aside from

assisting in the separation of statements into facts and goals as was just described, its presence serves as a cue for the program. Namely, the program interprets the statement A→B in a manner which is different from the way it reacts to the logically equivalent statement ~A∨B. The case analysis of ~A∨B involves a commitment by the program to prove A so that B can be concluded. However, rule R6 applied to A→B does not involve any commitment to prove A; it just says that if A already has been regarded as true, then we can conclude B. In particular, the program will not conclude C→B from the two statements C→A and A→B.

## 4. The Use of Procedures

While the program is not intended as a new programming language, it does have features which make it a meaningful extension of LISP. The theorem prover is a LISP function named DERIVEX which operates upon three arguments (1) the initial list of formulas stripped of all quantifiers, (2) a list of variables whose values are desired as the answer to the problem, and (3) NIL or T depending upon whether or not the request is for just one answer or for all possible answers. If it is desired that a particular formula A be an initial goal, then the formula would be coded as (GOALX A).

Whenever a literal P(T1,.....Tn) or a literal ~P(T1,.....Tn) has been established as true and is ready for processing by the deduction rules, its predicate letter P is examined to see whether it is also a LISP function. If the answer is yes, then P(T1,.....Tn) is evaluated and, depending

upon the output from this evaluation, there would be four possible responses: (1) if the ouput is T, the case under consideration is declared solved, (2) if the output is NIL, the literal is rejected and therefore would not be applied to the deduction rules, (3) if the output Q is a formula other than P(T1,.....Tn), then it replaces P(T1,.....Tn) and the resulting formula (i.e. either Q or ~Q depending upon whether the original literal was P(T1,.....Tn) or ~P(T1,.....Tn)) would await its turn for processing by the deduction rules, and (4) otherwise, the deduction continues as if the predicate letter P were not a LISP function. It should be noted with respect to the third possibility that the output Q does not have to be another atomic formula like P(T1,.....Tn); also, Q is regarded as a replacement for P(T1,.....Tn) and not as a descendant of P(T1,.....Tn).

Since the theorem prover DERIVEX is a LISP function, it can be used as a subroutine by other LISP functions. In particular, a LISP function that uses it might also be a predicate letter so that recursive applications of DERIVEX are possible. The theorem prover will examine function letters and will execute them if they are also LISP functions. The difference between the procedural treatment of function letters and predicate letters is that the former are executed immediately whereas the latter are executed only when appearing in a detached literal which is ready for processing by the deduction rules.

The formula A→B would be represented in LISP as (IMPLIES A B). However, in that form it never would be used as an input to any of the the modus ponens rules (i.e. R6, R7 or

R8). If one wanted to allow for a possible application to rule R6, then it would be coded as (IMPLIES (ROUTINEX A R) B) where R is some LISP function whose purpose is to screen out undesirable literals A' from application to rule R6. If one wanted to allow for a possible application of A→B to all three modus ponens rules, then it would have to be coded in the form (IMPLIES (ROUTINEX A R1) (ROUTINEX B R2)) where the LISP functions R1 and R2 would likewise help filter out unprofitable applications of these rules. The use of ROUTINEX is at the discretion of the human programmer. The filter would be disengaged by a choice of R1 and R2 which always evaluated as T. An example of its nontrivial use is given in section 5 together with other facilities for representing knowledge in the form of procedures.

## 5. The Relaxation Stage

Since the relaxation stage involves searching for "all" solutions to a case, one might object on the grounds that such a search could take too long (and in fact might never be completed). However, the search does not have to be for all possible solutions but rather for all solutions that can be obtained when certain heuristics are employed. Moreover, this search need not be conducted for every case generated by the split, as will be shown below. The degree of success should depend upon the problem domain and the individual heuristics. The program in [7] searched for "all" solutions to the first case of a split and yet performed quite creditably in domains such as number theory and group theory.

Suppose the program wishes to split the formula A(x)vB(x) into cases. Although all existential variables were removed at the outset by the skolemization process, the variable x, which appears in both cases, will be called (as in [7]) an "existential" variable since the object is to find some value of x which will permit a solution to each case. First, an attempt is made to find values of x which will solve the problem under the assumption that A(x) is true. After all the successful values of x are recorded, the same procedure is applied to B(x) with x still treated as a variable and not constrained only to those values that yielded solutions to the previous case A(x). Suppose that the formula B(y)→(C(y)vD(y)) is part of the data base for case B(x). Then, the application of B(y)→(C(y)vD(y)) with B(x) to rule R6 would match the variables x and y and generate the formula C(x)vD(x). The formula C(x)vD(x) would become a candidate for splitting within the scope of case B(x) and the program would not hesitate to attempt such a split (unless a solution of B(x) could be found which did not require the assignment of a specific value to some existential variable appearing in B(x)). However, suppose that instead of B(y)→(C(y)vD(y)), the formula B(y)→(C(y,z)vD(y,z)) had appeared in the data base where z is a variable that does not have an existential interpretation. The application of this formula with B(x) to rule R6 would generate the formula C(x,z)vD(x,z) but in order to split this latter formula an existential interpretation would have to be created for the variable z. Experience with [7] and the present program has indicated that it is usually not a

good idea to allow a split if (1) one or more existential variables appeared already in the hypothesis of the the case under consideration and (2) the new split would create one or more additional existential variables. So we did something to prevent this additional split. One method is to use the filter mechanism of section 4 by coding B(y)→(C(y,z)∨D(y,z)) as (IMPLIES (ROUTINEX B(y) R) (OR C(y,z) D(y,z))) where routine R would reject any formula that possessed an existential variable. Alternatively, there is a global variable which is zero or one depending upon whether or not a case analysis is being conducted that already has generated existential variables; therefore, the routine R might reject a formula if this global variable had the value zero. The program also would reject such a split automatically if there were nested above it a previous split which violated the above principle regarding the introduction of existential variables (i.e. even if the filter allowed the program to attempt the split of C(x,z)∨D(x,z) in order to solve one of the cases of A(x)∨B(x), it would never attempt a split of say E(w)∨F(w) involving another new existential variable w if this latter formula were generated from either case C(x,z) or case D(x,z)).

A LISP function called RECURSIVEX enables the human user to inform the program that a particular case should be postponed until after the rest of the split has been solved; when this case finally is attempted, it is with values that are known to satisfy the other cases of the split. Thus, a split of the formula A(x)∨RECURSIVEX(B(x))∨C(x) would result in formulas B(K1),

B(K2),.....B(Kn) being added to the data base where K1, K2,.....Kn represent the different solutions for x that were obtained from splitting A(x)vC(x). These formulas B(K1),.....,B(Kn) then would be available for use in an attempt to split some other formula D(y)vE(y).

The motivation behind the use of RECURSIVEX is that there might be semantic reasons for believing that a particular formula B(x) might be either (1) too "expansive" if one of its existential variables x is allowed to be unconstrained (i.e. the hypothesis B(x) might generate too many formulas if x is unrestricted) or (2) B(x) might lend itself to recursive use in a manner that would make the above heuristic governing the introduction of new existential variables appear too severe a restriction. If either of these conditions prevailed, it would be more appropriate to code B(x) as RECURSIVEX(B(x)).

Another LISP function called TESTX postpones action on a case until the reconciliation stage (as opposed to RECURSIVEX which postpones action until after the reconciliation stage is completed). By coding B(x) as TESTX(B(x)) the human user informs the program that B(x) is to be executed as a LISP function whenever a value is assigned to any existential variable x appearing in B(x); the assignment of such a value c to x would be rejected if the execution of B(c) returned a value of NIL. TESTX means the case is to be used to test rather than generate new assignments to existential variables. For example, x > 0 normally would be coded as TESTX(x > 0) since we do not wish to generate all the positive integers as values for x but only to

test whether a particular value is a positive integer.

## 6. The Reconciliation Stage

Throughout this section the word "variable" will refer to existential variable. The input to the reconciliation stage consists of a list of alternative solutions for each case. Each alternative solution to a case is represented as a list of attribute-value pairs where each attribute is either a variable, the symbol RECURSIVEX1 or the symbol TESTX1. If the attribute is a variable, then the value to which it is paired is its assignment for this particular solution. If the attribute is either RECURSIVEX1 or TESTX1, then it is paired with the hypothesis of a case still to be solved as it corresponds to the application of RECURSIVEX or TESTX respectively as described in section 5.

For each variable x, the program first constructs a constraint set of values consistent with the results of the relaxation stage. This constraint set is defined as the intersection of U(x,c) taken over all cases c where U(x,c) is the set of possible values for x as determined from the various solutions to case c. Thus, if a solution to case c did not impose any restriction on the variable x, then U(x,c) would be the universal set consisting of all elements. Also, U(x,c) would be taken as the universal set if a solution of x for case c depended upon another existential variable y (e.g. x = f(y) where f is a skolem function). Otherwise, U(x,c) is just the union of all values of x obtained from all the solutions to case c. By

computing the constraint set for each variable, the program is in a position to (1) abandon the reconciliation search if the constraint set for some variable is empty and (2) reject a newly computed value of a variable if this value is not consistent with at least one value from the constraint set of that variable.

Next, the program makes a determination as to the order in which cases are to be reconciled. First priority is given to cases with the fewest number of alternative solutions. In the event of a tie, the case is selected which involves the fewest number of "unbound" variables where a variable is said to be "bound" if its constraint set consists of only a single element.

Once a case is selected, its first alternative solution (consisting of a list of attribute-value pairs) is examined. Each attribute-value pair is processed in turn. Any variable appearing in the list structure formed by the attribute-value pair is replaced by its assigned value if such an assignment already had been made from some previous case. Let (A B) be the resulting attribute-value pair after these replacements (if any) have been made. If A is the symbol TESTX1, then the expression B would be evaluated and the current solution rejected should the evaluation return NIL. If A is neither the symbol TESTX1 nor RECURSIVEX1 and either (1) A is a variable which appears as a subelement of B, (2) B is a variable which appears as a subelement of A, or (3) neither A nor B are variables, then the theorem prover would attempt to prove A = B and would reject the current solution to the case if it failed in the attempt. If it succeeds in this attempt, it would return any values it might

have found for the variables in question and then would continue with the next attribute-value pair of the current solution.

Suppose A were a variable which was not a subelement of B. If B = B(x) depended upon exactly one variable x and the constraint set of x were finite, then each value k from the constraint set of x would be tested to see whether B(k) belonged to the constraint set of A. If no value B(k) so belonged, then the current solution would be rejected since it would be impossible for A to equal B. If exactly one B(k) belonged to the constraint set of A, then this (x k) would be treated as a new attribute-value pair (i.e. this k would be assigned as the value of variable x). The program then would proceed as if B did not depend upon just one variable. In particular, all previously processed attribute-value pairs would be examined to see whether they contained occurrences of the variable A. All such occurrences of A would be replaced by B and any pairs so involved would once again become candidates for processing in the same manner as has just been described.

If all the attribute-value pairs were processed successfully, then another case would be generated for reconciliation. A solution to the split would be obtained if no more cases remained to be reconciled. If (1) this split had only a single solution as its objective and (2) the solution that was just found did not necessitate the execution of another formula by virtue of a postponement through the use of RECURSIVEX, then an exit would be made from the reconciliation stage; otherwise, the program would record this solution (even if it includes a

formula whose execution has been deferred) and then would backtrack in a search for additional solutions.

When a solution to a case has been abandoned, the next alternative solution would be examined. If there are no more alternative solutions to a case, the program would backtrack to the previous case and examine its next alternative solution. An exit would be made from the reconciliation stage when the first case has no more alternative solutions to be examined.

## 7. Some Illustrative Examples

This section will present an in depth description of two examples (run on a PDP-10 computer) which should illustrate many of the features of the program. Although the program has had success with other problems, such as provided by the M.I.T. blocks world (e.g. in a matter of seconds it can find the largest red cube or all arches on the table from situations as complex as those considered in [13]), their description would not shed much more light on the way the program operates. Also, since the present program represents an improvement of an earlier theorem prover [7] which was successful in group theory and number theory, there is reason to believe that the present program would have as much success in these same domains if given a comparable equality rule to work with.

First, a few LISP functions that are used in these examples will be explained. MINUSX, PLUSX, and TIMESX are arithmetic functions which correspond to subtraction, addition, and multiplication respectively. MINUSX is a function of two

arguments defined by (MINUSX x y) equals (PLUSX x (TIMESX -1 y)). PLUSX and TIMESX are functions of an indefinite number of arguments whose purpose is to carry out addition and multiplication in the presence of variables to which numerical values may not yet have been assigned. Thus, (PLUSX 3 -2 4) returns the value 5 but (PLUSX 3 x 4) would return the value (PLUSX x 7) if no numerical value had been assigned to x. The expression
(TIMESX 3 (PLUSX (TIMESX 2 x) y 4 (MINUSX z (TIMESX 2 x)))) when evaluated would return (PLUSX (TIMESX 3 y) (TIMESX 3 z) 12). GEX is a function of two arguments. (GEX x y) returns (GEX x y) if either x or y does not have a numerical value. Otherwise, (GEX x y) returns T or NIL depending upon whether or not x is as large a number as y.

### 7.1 Missionaries and Cannibals

This task (3) has three missionaries and three cannibals who wish to cross a river from the left to the right side. Their only means of conveyance is a boat which has a capacity of two people. Any of the missionaries or cannibals is capable of operating the boat either alone or with someone else. The problem is to determine a way by which all the missionaries and cannibals can be transported to the right side without ever allowing the cannibals to outnumber the missionaries on any one side since otherwise the missionaries on that side would be eaten.

Let (LEFT cx my) represent the assertion that it is possible to transport all cannibals and missionaries to the right

side given that the boat is on the left side together with cx cannibals and my missionaries. Let (RIGHT cx my) represent the assertion that it is possible to transport all missionaries and cannibals to the right side given that the boat is on the right side together with cx cannibals and my missionaries. The initial data base for this problem consists of the following three formulas where x and y should be interpreted as referring to the number of cannibals and missionaries respectively being sent across in the boat:

1. (GOALX (NOT (LEFT 3 3))) which is the initial goal of proving (LEFT 3 3).

```
2.  (IMPLIES (AND (TESTX (GEX cx x))
                  (TESTX (GEX my y))
                  (OR (EQUALS x 0) (EQUALS x 1) (EQUALS x 2))
                  (OR (EQUALS y 0) (EQUALS y 1) (EQUALS y 2))
                  (OR (EQUALS x (MINUSX 1 y)) (EQUALS x (MINUSX 2 y)))
                  (OR (AND (EQUALS cx x) (EQUALS my y))
                      (AND (OR (EQUALS x (PLUSX cx (MINUSX y my)))
                               (EQUALS my y)
                               (EQUALS y (MINUSX my 3)))
                           (RECURSIVEX (RIGHT (MINUSX 3 (MINUSX cx x))
                                              (MINUSX 3 (MINUSX my y))
                                                                  )))))
           (ROUTINEX (LEFT cx my) TRUEX))

3.  (IMPLIES (AND (TESTX (GEX cx x))
                  (TESTX (GEX my y))
                  (OR (EQUALS x 0) (EQUALS x 1) (EQUALS x 2))
                  (OR (EQUALS y 0) (EQUALS y 1) (EQUALS y 2))
                  (OR (EQUALS x (MINUSX 1 y)) (EQUALS x (MINUSX 2 y)))
                  (OR (EQUALSX x (PLUSX cx (MINUSX y my)))
                      (EQUALS my y)
                      (EQUALS y (MINUSX my 3)))
                  (RECURSIVEX (LEFT (MINUSX 3 (MINUSX cx x))
                                    (MINUSX 3 (MINUSX my y)))))
           (ROUTINEX (RIGHT cx my) TRUEX))
```

TRUEX is a function of one argument which always returns a value of T; its purpose is to disengage the modus ponens filter. The statement (GEX cx x) says that the number of

cannibals to be sent in the boat cannot exceed the number available on that side of the river; however, by coding this statement as (TESTX (GEX cx x)) the human programmer has directly informed the program that (GEX cx x) is to be under the control of TESTX as described in section 5. The statement (OR (EQUALS x (MINUSX 1 y)) (EQUALS x (MINUSX 2 y))) says that the combined number of cannibals and missionaries to be sent in the boat must be either 1 or 2. The satisfaction of (AND (EQUALS cx x) (EQUALS my y)) in formula 2 means that the problem would be solved since all remaining missionaries and cannibals would be in the boat and on their way to the right side of the river. The statement (OR (EQUALS x (PLUSX cx (MINUSX y my))) (EQUALS my y) (EQUALS y (MINUSX my 3))) says that either an equal number of cannibals and missionaries must be left behind on shore after the boat departs (i.e. cx - x = my - y) or the number of missionaries being left behind (i.e. my - y) is either 0 or 3; this expresses the requirement that missionaries cannot be in the presence of a greater number of cannibals. The statement (RIGHT (MINUSX 3 (MINUSX cx x)) (MINUSX 3 (MINUSX my y))) says that the problem now must be solved from a starting point which has the boat together with 3 - (cx - x) cannibals and 3 - (my - y) misssionaries on the right side of the river; by coding this statement with RECURSIVEX, the human programmer has directly informed the program that the statement is to be under the control of RECURSIVEX as described in section 5.

The program initially pairs formulas 1 and 2 to rule R7 for cx = my = 3 and then, after applying the output to rule R10,

it attempts to split the resulting formula. This split results in the generation of the three additional formulas (NOT (RIGHT 1 0)), (NOT (RIGHT 2 0)), and (NOT (RIGHT 1 1)) which in turn become candidates for application to the deduction rules. The processing of (NOT (RIGHT 1 0)) generates (NOT (LEFT 3 3)) which is rejected since it is an instance of a previous formula. The processing of (NOT (RIGHT 2 0)) generates (NOT (LEFT 3 3)) (which again is rejected) and also generates (NOT (LEFT 2 3)). The program continues in this manner and obtains a proof in three minutes. In evaluating this performance, it is well to keep in mind that the missionaries and cannibals problem lends itself more to a representation based upon GPS type operators [9] rather than upon the predicates of automatic theorem provers.

## 7.2 Cryptarithmetic

This task [9], [3] involves the assignment of a decimal digit to each of the letters of three words so that the sum of the first two words equals the third word. No digit may be assigned to more than one letter and no word may have zero assigned to its first letter.

As a sample problem, consider DONALD + GERALD = ROBERT [9]. Although the program solved this problem without hints, it will be more illustrative to examine how the program attacked this problem when provided with the same hint of D = 5 that had been given elsewhere [9] to human subjects. This problem is represented by the following formula where K1 through K5 are the "carries" from one column to the next which the program must determine.

```
1.  (GOALX (NOT (AND
        (OR (AND (EQUALS (PLUSX D D) T) (EQUALS K5 0))
            (AND (EQUALS (PLUSX D D) (PLUSX 10 T)) (EQUALS K5 1)))
        (OR (AND (EQUALS (PLUSX L L K5) R) (EQUALS K4 0))
            (AND (EQUALS (PLUSX L L K5) (PLUSX 10 R)) (EQUALS K4 1)))
        (OR (AND (EQUALS (PLUSX A A K4) E) (EQUALS K3 0))
            (AND (EQUALS (PLUSX A A K4) (PLUSX 10 E)) (EQUALS K3 1)))
        (OR (AND (EQUALS (PLUSX N R K3) B) (EQUALS K2 0))
            (AND (EQUALS (PLUSX N R K3) (PLUX 10 B)) (EQUALS K2 1)))
        (OR (AND (EQUALS (PLUSX O E K2) O) (EQUALS K1 0))
            (AND (EQUALS (PLUSX O E K2) (PLUSX 10 O)) (EQUALS K1 1)))
        (EQUALS (PLUSX D G K1) R)
        (EQUALS D 5)
        (GENERATORX A 0 9)
        (GENERATORX B 0 9)
        (GENERATORX D 1 9)
        (GENERATORX E 0 9)
        (GENERATORX G 1 9)
        (GENERATORX L 0 9)
        (GENERATORX N 0 9)
        (GENERATORX O 0 9)
        (GENERATORX R 1 9)
        (GENERATORX T 0 9)
        (TESTX (DISTINCTX A B D E G L N O R T)))))
```

The statement (DISTINCTX A B D E G L N O R T) in formula 1 reflects the requirement that distinct digits must be assigned to distinct letters (i.e. DISTINCTX is a function of an indefinite number of arguments which returns a value of NIL or T depending upon whether or not two of its arguments have identical values). The ability to handle linear equations (by solving for one variable in terms of the others) was built into the predicate EQUALS. Also, the expressions of the form (GENERATORX x y z) serve to generate alternative solutions to a case by assigning integer values to variable x with the values ranging from y to z. The LISP function GENERATORX was used only to make the problem statement more compact. The problem was solved at about the same speed when statements like (GENERATORX A 0 9) were replaced by (OR (EQUALS A 0) (EQUALS A 1) ..... (EQUALS A 9)).

The program is asked to find values for the variables A,

B, D, E, G, L, N, O, R and T that will result in the attainment of the goal). It begins by applying formula 1 to rule R10 and then attempts to split the resulting formula. After completing the relaxation stage (i.e. solving the different cases independently), it constructs a constraint set for each variable and determines an order in which the different cases are to be reconciled. The following description of the program's attempt to reconcile the different solutions invites comparison with human protocols recorded in [9]. Although some amount of brute force is intrinsic to the problem, the program is able to use its knowledge about the nature of these solutions (e.g. they are distinct integers constrained by the results of the relaxation stage) to guide the search in an effective manner.

First, it concludes D = 5 and R = D + G + K1 since these were the only solutions generated by their respective cases. Therefore, R = 5 + G + K1. It then selects the case that had generated the two alternative solutions (1) T = 2*D, K5 = 0 and (2) T = 2*D - 10, K5 = 1. The first alternative solution T = 2*D, K5 = 0 is rejected since it would imply T = 2*5 = 10 which does not fall within the constraint set for T (i.e. one of the integers 0 through 9). Therefore, K5 = 1 and T = 2*D - 10 = 2*5 - 10 = 0. The next case selected had generated the two alternative solutions (1) K1 = 0, E = -K2 and (2) K1 = 1, E = 10 - K2. It first assumes K1 = 0, E = -K2 and notices that E is expressed in terms of a single variable K2. It therefore looks at the constraint set of K2 which consists of only the values 0 and 1. However, it rejects the value K2 = 1 since that would

imply E = -1 which falls outside the constraint set for E. It therefore concludes that K2 = 0. But this too is quickly rejected since it would imply E = 0 = T which violates the condition (activated by the LISP function DISTINCTX) that no two letters may have the same decimal digits. Therefore, K1 = 1, E = 10 - K2 and hence R = 5 + G + K1 = G + 6. Once again, it looks at the constraint set for K2 and rejects the value K2 = 0 since it would imply E = 10 - K2 = 10 which falls outside the constraint set for E. Therefore, K2 = 1 and hence E = 10 - K2 = 9.

The next case selected had generated the two alternative solutions (1) K3 = 0, E = 2*A + K4 and (2) K3 =1, E = 2*A + K4 - 10. It first assumes K3 = 0, E = 2*A + K4 and hence 2*A + K4 = 9 since E = 9. The theorem prover then proves 2*A + K4 = 9 by solving this linear equation for the variable A. It gets A = 4.5 - .5*K4 which suggests that it examine the constraint set of K4 since A is expressed in terms of the single variable K4. The value K4 = 0 is rejected since A = 4.5 then would fall outside the constraint set for A. Therefore, K4 = 1 and hence A = 4. It next selects the case that had yielded the two solutions (1) K4 = 0, R = 2*L + K5 and (2) K4 = 1, R = 2*L + K5 -10 and rejects the first alternative since it contradicts K4 = 1. Therefore, R = 2*L + K5 -10. However, since it already has deduced K5 = 1 and R = G + 6, it concludes G + 6 = 2*L - 9 which when solved for L gives L = 7.5 + .5*G. It then selects the case that had yielded the two solutions (1) K2 = 0, B = N + R + K3 and (2) K2 = 1, B = N + R + K3 - 10 and rejects the first alternative since it

contradicts K2 = 1. Therefore, B = N + R + K3 - 10. However, since it already believes K3 = 0 and R = G + 6, it gets B = N + G - 4. Next, the program examines the case that had produced the integers 1 through 9 as possible values for R. The value R = 9 is rejected since this value already was assigned to E. The value R = 8 is quickly rejected since it results in G = R - 6 = 2 and hence L = 7.5 + .5*G = 8.5 which falls outside the constraint set of L. However, the choice R = 7 is accepted as it results in G = 1, L = 8, and B = N + G - 4 = N - 3. The program continues in this manner and eventually finds O = 2, N = 6, and B = 3 which solves the problem. The total elapsed time was 48 seconds. It took the program five minutes to complete the more difficult problem of finding the solution without the hint of D = 5.

## References

1. Bledsoe, W.W., Splitting and reduction heuristics in automatic theorem proving, Artificial Intelligence, Vol. 2, (Spring 1971), pp. 55-77.

2. Bledsoe, W.W., Boyer, R.S., and Henneman, W.H., Computer proofs of limit theorems, Artificial Intelligence, Vol. 3, (Spring 1972), pp. 27-60.

3. Fikes, R.E., REF-ARF: A system for solving problems stated as procedures, Artificial Intelligence, Vol. 1, (Spring 1970), pp. 27-120.

4. Hewitt, C., Planner: A language for proving theorems in robots, Proc. Int. Joint Conf. on Artificial Intelligence, Washington, D.C., 1969, pp. 295-301.

5. Huet, G.P., Experiments with an interactive program for logic with equality, Report No. 1106, Jennings Computing Center, Case Western Reserve University, 1971.

6. McDermott, D.V, and Sussman, G.J., The Conniver reference manual, A.I. Memo No. 259, Massachusetts Institute of Technology, May 1972.

7. Nevins, A.J., A human oriented logic for automatic theorem proving, J. Assoc. Computing Machinery (forthcoming).

8. Nevins, A.J., Plane geometry theorem proving using forward chaining, A.I. Memo No. 303, Massachusetts Institute of Technology, January 1974.

9. Newell, A. and Simon, H.A. Human Problem Solving, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

10. Nilsson, N.J., Problem Solving Methods in Artificial Intelligence, McGraw Hill, New York City, N.Y., 1971.

11. Norton, L.M. Experiments with a heuristic theorem-proving program, Artificial Intelligence, Vol. 2, (Winter 1971), pp. 261-284.

12. Sussman, G.J. A Computational Model of Skill Acquisition, Ph.D. thesis, Massachusetts Institute of Technology, August 1973.

13. Winograd, T. Understanding Natural Language, Academic Press, New York City, N.Y., 1972.

14. Wos, L., Robinson, G.A., and Carson, D.F., Efficiency and completeness in the set of support strategy in theorem proving, J. Assoc. Computing Machinery, Vol. 12, No. 4, (October 1965), pp. 536-541